Permutations d'un ensemble fini
(Programmation avec Maple)

Préparation à la nouvelle épreuve d'informatique
de l'École Polytechnique.

Jean-Michel Ferrard

Avant-propos

Ce document vise à la préparation à la nouvelle épreuve d'informatique de l'École Polytechnique.

Les caractéristiques de cette épreuve peuvent être consultées à l'adresse :

http ://www.enseignement.polytechnique.fr/informatique/concours/

Parmi les langages de programmation possibles, on a choisi Maple, qui est le plus connu actuellement des élèves de classe préparatoire n'ayant pas suivi l'option informatique.

Les concepteurs de l'épreuve recommandent de n'utiliser que des fonctionnalités de base, de façon à travailler sur le plus petit dénominateur commun à tous les langages autorisés.

Dans ce document, on a scrupuleusement respecté ces consignes d'économie, même si c'est à regret. On n'ignore pas que de nombreuses fonctionnalités de Maple permettent de réécrire complètement certaines des fonctions ou procédures qui seront présentées dans ces pages.

Pour ne prendre qu'un exemple, la fonction `identp` de la question I-1 (et dont le rôle est de former le tableau associé à la permutation identité de $\mathcal{S}_n$) pourrait s'écrire :

```
identp:=(n::posint)->array([$1..n])
```

Le thème du présent document est l'étude du groupe symétrique $\mathcal{S}_n$, c'est-à-dire l'ensemble des bijections de $E_n = \{1, 2, \ldots, n\}$ dans lui-même.
On a inclus quelques rappels de cours, qui sont plus que des rappels pour la filière PC* (où le groupe symétrique et la signature sont hors-programme) et dans certains cas pour la filière PSI* (en ce qui concerne les propriétés des cycles.)

Voici rapidement quelques remarques utiles sur le contenu de ce document :

- La structure de liste n'étant pas autorisée, on utilisera des objets de type `array`.
- Les programmes (procédures et fonctions) seront toujours écrits avec la syntaxe `proc(...)...end`.
- On distinguera bien les *procédures* des *fonctions*.
  Les deux structures reçoivent des *arguments* nécessaires à leur fonctionnement, mais :
  - ◇ Une fonction *renvoie* un *résultat*, directement utilisable dans une *expression*.
    Par exemple, on écrira `P:=identp(10)` pour mettre l'identité de $\mathcal{S}_{10}$ dans la variable P.
    Une fonction ne modifie pas les données qui lui sont extérieures (i.e. les variables *globales*.)
  - ◇ Une procédure ne renvoie aucun résultat, mais peut modifier le contenu des variables globales.
    Si on écrit par exemple `identp` comme une procédure (recevant en argument un tableau et un entier), l'*instruction* `identp(P,10)` aura pour effet de modifier le contenu de la variable P.
  - ◇ On utilisera toujours `RETURN()` pour forcer une procédure à ne renvoyer aucun résultat.

  - ◇ On utilisera toujours `RETURN(...)` pour préciser le résultat renvoyé par une fonction (bien qu'avec Maple ce soit souvent inutile, le résultat étant celui de la dernière expression évaluée.)
- Pour gagner un peu en lisibilité, les identificateurs des procédures commenceront par une majuscule et ceux des fonctions par une minuscule. On ne "typera" pas les variables locales. On désignera seulement les "`integer`" par une lettre minuscule et les "`array`" par une lettre majuscule.
- Quand une procédure ou fonction utilise une procédure ou fonction précédemment écrite, l'identificateur de cette dernière est signalé en caractères italiques dans le texte.

Table des matières

Énoncé du problème

Voir en fin de document, pour des rappels de cours

Pour tout entier $n \geq 1$, on note $\mathcal{S}_n$ le groupe des permutations de $E_n = \{1, 2, \ldots, n\}$.

Un élément p de $\mathcal{S}_n$ sera représenté par $P = [p(1), p(2), \cdots, p(n)]$, tableau indicé de 1 à n.

Par exemple $P = [5, 3, 1, 4, 6, 2]$ est l'élément p de $\mathcal{S}_6$ défini par $\begin{cases} p(1) = 5,\ p(2) = 3,\ p(3) = 1 \\ p(4) = 4,\ p(5) = 6,\ p(6) = 2 \end{cases}$

Dans toute la suite, on sera amené à programmer puis à utiliser des *fonctions* et des *procédures* agissant sur des permutations p (i.e. sur les tableaux P associés.) On supposera que tout tableau transmis en argument est valide. Plus généralement, on ne cherchera pas à vérifier que les arguments reçus par une fonction ou une procédure sont conformes à ce qui est attendu.

On pourra cependant procéder à un typage minimal des arguments, en se limitant aux types `integer` (entier relatif) et `array(integer)` (tableau d'entiers).

Bien sûr, quand une fonction est censée renvoyer une permutation (i.e. le tableau associé), il nous appartiendra de faire en sorte que ce tableau soit valide !

Rappels

- L'expression `array(1..n)` renvoie un tableau indicé de 1 à n.
- L'expression `rand()` renvoie un entier pseudo-aléatoire à douze chiffres.
- Pour tout réel x, l'expression `floor(`x`)` renvoie l'entier "partie entière" de x.
- Pour tous entiers a et $b > 0$, a `mod` b ou `irem(`a, b`)` renvoient le reste dans la division de a par b.
  Les expressions `iquo(`a, b`)` ou `floor(`a/b`)` donnent le quotient entier de cette division.

I. Opérations sur les permutations

1. Écrire une fonction `identp` créant la permutation identité dans $\mathcal{S}_n$.
   L'argument est un entier $n \geq 1$, et le résultat est le tableau $[1, 2, \ldots, n]$. [S]
2. Écrire une fonction `randp` créant une permutation p pseudo-aléatoire de $\mathcal{S}_n$.
   L'argument est un entier $n \geq 1$, et le résultat est un tableau $P = [p(1), p(2), \ldots, p(n)]$. [S]
3. Écrire une fonction `comp` composant deux permutations de $\mathcal{S}_n$.
   La syntaxe d'appel est `comp(`q, p, n`)` où n est dans $\mathbb{N}^*$ et q, p dans $\mathcal{S}_n$.
   Le résultat renvoyé est la permutation $q \circ p$ (i.e. le tableau associé.)
   Dans `comp`, on ne vérifiera pas que p et q désignent effectivement des éléments de $\mathcal{S}_n$. [S]
4. On se propose d'écrire une fonction `powp` calculant $q = p^m$, où p est dans $\mathcal{S}_n$ et m dans $\mathbb{N}$.
   Avec la syntaxe d'appel `powp(`p, n, m`)`, le résultat est le tableau associé à $q = p^m$.
   On donnera deux solutions, utilisant toutes deux un algorithme d'exponentiation rapide.
   (a) La première solution sera itérative (non récursive). [S]

(b) La deuxième solution sera récursive. [S]

5. Écrire une fonction `invp` calculant l'inverse d'une permutation p de $\mathcal{S}_n$.
Avec la syntaxe `invp(`p, n`)`, le résultat est le tableau Q associé à $q = p^{-1}$.
On commencera par créer un tableau Q de longueur n, dont on affectera les différentes composantes pour en faire le tableau associé à p^{-1}. [S]

II. Décomposition en cycles disjoints

1. Écrire une *procédure* inversant une permutation p de $\mathcal{S}_n$ "sur place", donc sans utiliser (comme dans I-5) un tableau auxiliaire de la taille de celui associé à p.
La syntaxe sera `Invp(`P, n`)`. `Invp` ne renvoie aucun résultat (ce n'est pas une *fonction*).
Elle transforme le tableau P passé en argument en celui associé à la permutation inverse.
[Indication] [S]

2. Écrire une *procédure* transformant une permutation en sa décomposition en produits de cycles, de la manière qui est décrite dans l'exemple suivant :
Considérons $p = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 7 & 2 & 5 & 8 & 1 & 4 & 3 & 6 \end{pmatrix} = (1 \quad 7 \quad 3 \quad 5) \circ (4 \quad 8 \quad 6)$.
Dans cet exemple `decomp(`$p, 8$`)` transformera p en $[1, 7, 3, -5, -2, 4, 8, -6]$.
On devra donc former un tableau de même longueur que le tableau initial, faisant apparaître les cycles successifs. *La fin d'un cycle est marquée par un changement de signe.*
Un point fixe étant un cycle de longueur 1, il est lui même affecté d'un changement de signe.
Avec ces conventions, $[1, 7, 3, -5, -2, 4, 8, -6]$, $[4, 8, -6, 1, 7, 3, -5, -2]$, $[7, 3, 5, -1, -2, 6, 4, -8]$, etc. sont des représentations valides de la décomposition d'une même permutation p.
On dira qu'une permutation p de $\mathcal{S}_n$ est "sous forme $\mathcal{C}$" si le tableau qui la représente (et qui est lui-même indicé de 1 à n) est construit avec les conventions indiquées précédemment.
Il n'y a évidemment pas unicité de la forme $\mathcal{C}$ d'une permutation p.
On dira que $[p(1), p(2), \ldots, p(n)]$ est la forme $\mathcal{T}$ de la permutation p.
La procédure `Decomp` transforme donc la forme $\mathcal{T}$ de p en l'une de ses formes $\mathcal{C}$. [S]

3. Écrire une *procédure* `Recomp` effectuant le travail inverse de `Decomp`.
La procédure `Recomp` transforme donc une forme $\mathcal{C}$ d'une permutation en sa forme $\mathcal{T}$. [S]

4. Dans cette question, on va calculer de deux manières différentes la signature d'une permutation.

(a) Écrire une fonction `signat` calculant la signature d'une permutation p de $\mathcal{S}_n$.
La syntaxe sera `signat(`P, n`)` et le résultat sera $\varepsilon(p)$.
On suppose ici que le tableau P est la forme usuelle (forme $\mathcal{T}$) de p.
On utilisera une méthode naïve consistant à compter toutes les inversions de p. [S]

(b) Réécrire la fonction `signat` en supposant que le tableau P transmis en argument est une représentation sous forme $\mathcal{C}$ (comme fournie par `Decomp`) de la procédure p. [S]

5. Écrire une fonction `ordre` calculant l'ordre d'une permutation p (représentée par le tableau P de sa décomposition en cycles, à la manière de celui obtenu par la procédure `Decomp`.) [S]

6. En utilisant la décomposition en cycles fournie par la procédure `Decomp`, écrire une *procédure* `Powp` calculant une puissance m quelconque (y compris négative) d'une permutation p de $\mathcal{S}_n$.
   La syntaxe d'appel sera `Powp(P,n,m)`, où P est le tableau représentant p sous sa forme "traditionnelle" (c'est-à-dire sous la forme $\mathcal{T}$).
   La procédure `Powp` ne renverra aucun résultat mais modifiera le tableau P pour en faire celui associé à la permutation p^m (toujours sous la forme $\mathcal{T}$.)
   [Indication] [S]

III. Génération lexicographique de permutations

On se propose d'étudier une méthode "lexicographique" pour lister les éléments de $\mathcal{S}_n$.
Une permutation p est ici représentée par le tableau $P = [p(1), p(2), \ldots, p(n)]$.
L'ensemble $\mathcal{S}_n$ est totalement ordonné par l'ordre lexicographique.
Pour cet ordre, le minimum de $\mathcal{S}_n$ est $[1, 2, \ldots, n\text{-}1, n]$ et le maximum est $[n, n\text{-}1, \ldots, 2, 1]$.

1. Écrire une *procédure* `Nextp` transformant un tableau P (donc une permutation p de $\mathcal{S}_n$) en le tableau qui représente la permutation suivante q.
   La syntaxe est `Nextp(`P, n`)` et aucun résultat n'est renvoyé.
   Si p est la permutation maximum de $\mathcal{S}_n$, le tableau P n'est pas modifié.
   [Indication] [S]

2. Écrire une *fonction* `listp` donnant les k permutations qui viennent à partir d'une permutation p dans l'ordre lexicographique de $\mathcal{S}_n$. La syntaxe d'appel sera `Listp(P,n,k)` et le résultat sera un tableau de k lignes sur n colonnes (chaque ligne représentant une permutation, la première ligne correspondant à la permutation de départ p.)
   En déduire une fonction `listallp` renvoyant le tableau de toutes les permutations de $\mathcal{S}_n$. [S]

3. La méthode précédente permet d'ordonner complètement les éléments de $\mathcal{S}_n$.
   Chaque permutation reçoit donc un numéro d'ordre unique, de $m = 0$ pour $P = [1, 2, \ldots, n]$ (la permutation identité), à $m = n! - 1$ pour $P = [n, \ldots, 2, 1]$.

   Il est intéressant de pouvoir extraire une permutation à partir d'un numéro d'ordre, ou au contraire de connaître le numéro d'ordre d'une permutation donnée.

   Cela est possible grâce à une méthode appelée **codage de Lehmer**.

   Définition *(Code de Lehmer d'une permutation)*

   > Soit p une permutation de $\mathcal{S}_n$, avec $n \geq 2$.
   > Le *code de Lehmer* de p est n-uplet $\mathcal{L}(p) = (c_1, c_2, \ldots, c_{n-1}, c_n)$ défini par :
   > Pour tout i de $\{1, \ldots, n\}$, c_i est le nombre d'indices j de $\{i\text{+}1, \ldots, n\}$ tels que $p(j) < p(i)$.
   > Chaque c_i est donc le nombre d'inversions (i, j) de p, avec $i < j$. Ainsi $0 \leq c_i \leq n\text{-}i$.
   > Par construction $\mathcal{L}(p)$ est dans $F_n = \{0, \ldots, n\text{-}1\} \times \{0, \ldots, n\text{-}2\} \times \cdots \times \{0, 1\} \times \{0\}$.

Question : écrire une fonction `code` donnant le code de Lehmer d'une permutation p.
La syntaxe d'appel sera `code(P,n)` et le résultat sera le tableau $[c_1, c_2, \dots, c_n]$. [S]

4. Avec les notations précédentes, on remarque que pour tout i de $\{1, \dots, n\}$, on a $c_i < p(i)$.
C'est en effet une conséquence des conditions $p(j) < p(i)$ dans la définition.
Question : écrire une *procédure* `Code` transformant le tableau P représentant une permutation de $\mathcal{S}_n$ en le tableau représentant son code de Lehmer (la syntaxe est `Code(P,n)`.)
Pour cela, on imaginera une méthode de "descente" de $[p(1), \dots, p(n)]$ vers $[c_1, \dots, c_n]$.
[Indication] [S]

5. En cherchant à inverser la méthode précédente, écrire une procédure `Decode` permettant de reformer le tableau associé à une permutation p, à partir de son code de Lehmer $P = \mathcal{L}(p)$. La syntaxe est `Decode(P,n)`.
Comme d'habitude, on ne vérifiera pas que le tableau $P = [c_1, c_2, \dots, c_n]$ donné en argument est "valide", c'est-à-dire que chaque c_k est un entier de $\{0, \dots, n-k\}$. [S]

6. Soit p une permutation de $\mathcal{S}_n$, représentée par le tableau $P = [p(1), p(2), \dots, p(n)]$.
Soit $\mathcal{L}(p) = [c_1, c_2, \dots, c_{n-1}, c_n = 0]$ le code de Lehmer de p.
On lui associe le "rang" de p, défini par :
$$\mathrm{rg}(p) = c_1(n-1)! + c_2(n-2)! + \cdots + c_{n-1}1! + c_n 0! = \sum_{k=1}^{n} c_k (n-k)!$$
On verra l'interprétation de cette fonction dans la question suivante.
On sait que chaque entier c_k vérifie $0 \leq c_k \leq n-k$.
On remarque aussi que $\sum_{k=1}^{n} (n-k)(n-k)! = \sum_{k=1}^{n} ((n+1-k)!-(n-k)!) = n!-1$.
On définit ainsi une application $p \mapsto \mathrm{rg}(p)$ de $\mathcal{S}_n$ dans $\{0, 1, \dots, n!-1\}$.
Question : écrire une *fonction* calculant le rang d'une permutation p.
La syntaxe sera `rang(P,n)`, où $P = [p(1), p(2), \dots, p(n)]$.
NB : pour calculer $\mathrm{rg}(p)$, on n'utilisera pas la fonction factorielle... [S]

7. On rappelle la notation $F_n = \{0, \dots, n-1\} \times \{0, \dots, n-2\} \times \cdots \times \{0, 1\} \times \{0\}$.
L'ensemble F_n, qui est de cardinal $n!$, est totalement ordonné par l'ordre lexicographique.
   - Montrer que le code de Lehmer définit une application strictement croissante de $\mathcal{S}_n$ sur F_n.
   - Montrer que l'application $p \mapsto \mathrm{rg}(p)$ est strictement croissante de $\mathcal{S}_n$ sur $\{0, \dots, n!-1\}$.
   Elle permet donc de "numéroter" les éléments de $\mathcal{S}_n$ dans l'ordre lexicographique.
   - En inversant cette application, on retrouve une permutation à partir de son numéro d'ordre.
   Écrire une fonction `gnar` (l'inverse de la fonction `rang`) qui donne p dans $\mathcal{S}_n$ à partir de son rang r. La syntaxe sera `gnar(n,r)` et le résultat sera le tableau $[p(1), p(2), \dots, p(n)]$.
   Le point essentiel est bien sûr l'algorithme qui permet de retrouver le tableau $[c_1, c_2, \dots, c_n]$ (le code de Lehmer), à partir de $r = \sum_{k=1}^{n} c_k (n-k)!$
   - En déduire une nouvelle fonction `randp` de calcul d'une permutation pseudo-aléatoire de $\mathcal{S}_n$.
   [S]

IV. Génération récursive de permutations

Dans cette partie, on voit deux méthodes pour former toutes les permutations p de l'ensemble $\mathcal{S}_n$.

Les deux méthodes décrites ici utilisent une approche récursive.

1. Écrire une procédure `Printp` qui affiche successivement tous les éléments de $\mathcal{S}_n$.
   On respectera la contrainte suivante (que l'on considèrera comme une indication de l'algorithme à utiliser) : si p et q sont deux éléments de $\mathcal{S}_n$, la permutation p sera affichée avant la permutation q si p^{-1} est située avant q^{-1} dans l'ordre lexicographique.
   [Indication] [S]

2. Dans cette question, on engendre récursivement $\mathcal{S}_n$ par une méthode d'échanges successifs.
   (a) On suppose que l'entier n est supérieur ou égal à 2.
   Pour tout k de $E_n = \{1, \ldots, n\}$, on note τ_k la transposition (k, n) (avec $\tau = \mathrm{id}$ si $k = n$.)
   Montrer que l'application $\varphi : (\sigma, k) \mapsto \sigma \circ \tau_k$ est une bijection de $\mathcal{S}_{n-1} \times E_n$ sur $\mathcal{S}_n$. [S]

   (b) En déduire une nouvelle procédure `Printp` affichant tous les éléments de $\mathcal{S}_n$.
   On fera en sorte qu'apparaissent d'abord les permutations p telles que $p(n) = n$, puis celles telles que $p(n) = n - 1$, etc., jusqu'à celles telles que $p(n) = 1$. [S]

V. Permutations ayant un nombre donné de cycles

On sait que tout élément p de $\mathcal{S}_n$ peut être décomposé en un produit de cycles à supports disjoints (et d'une manière unique, à l'ordre près des facteurs.) Dans cette partie, on considérera qu'un point fixe d'une permutation en est un cycle de longueur 1. Les supports des différents cycles intervenant dans une permutation p forment donc une partition de l'ensemble $E_n = \{1, 2, \ldots, n\}$.

Ainsi $p = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 10 & 5 & 9 & 4 & 14 & 3 & 1 & 15 & 12 & 7 & 13 & 6 & 2 & 8 & 11 \end{pmatrix}$ se décompose en

$p = (1\ \ 10\ \ 7) \circ (2\ \ 5\ \ 14\ \ 8\ \ 15\ \ 11\ \ 13) \circ (3\ \ 9\ \ 12\ \ 6) \circ (4).$

On dira que cette permutation possède quatre cycles, de longueurs 1, 3, 4 et 7.

Tout élément p de $\mathcal{S}_n$ a donc k cycles, avec $1 \leq k \leq n$. La somme des longueurs de ces cycles vaut n.
Pour $1 \leq k \leq n$, on note $\begin{bmatrix} n \\ k \end{bmatrix}$ le nombre de permutations dans $\mathcal{S}_n$ qui ont exactement k cycles.
Les coefficients $\begin{bmatrix} n \\ k \end{bmatrix}$ sont appelés *nombres de Stirling de première espèce*.

1. Dans cette question, on établit la relation qui permet de calculer les $\begin{bmatrix} n \\ k \end{bmatrix}$ de proche en proche.
   (a) Préciser les valeurs de $\begin{bmatrix} n \\ n \end{bmatrix}$, de $\begin{bmatrix} n \\ n-1 \end{bmatrix}$ et de $\begin{bmatrix} n \\ 1 \end{bmatrix}$. [S]

   (b) Prouver que $\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix}$ (si $1 < k < n$.) En déduire une sorte de triangle de Stirling permettant de calculer les coefficients $\begin{bmatrix} n \\ k \end{bmatrix}$, avec par exemple $1 \leq k \leq n \leq 7$. [S]

2. Déduire de la question précédente une fonction `stir` permettant de calculer les $\begin{bmatrix} n \\ k \end{bmatrix}$.
   Avec la syntaxe `stir(`n`)` le résultat sera le tableau $\left[\begin{bmatrix} n \\ 1 \end{bmatrix}, \begin{bmatrix} n \\ 2 \end{bmatrix}, \ldots, \begin{bmatrix} n \\ n \end{bmatrix}\right]$.
   NB : on utilisera un algorithme itératif, et un seul tableau (unidimensionnel de taille n.) [S]
3. Écrire une fonction `randc` renvoyant une permutation aléatoire p de $\mathcal{S}_n$ ayant k cycles.
   Les différentes permutations solutions doivent pouvoir apparaître de façon équiprobable.
   La syntaxe d'appel sera `randc(`n, k`)` et le résultat sera le tableau $[p(1), p(2), \ldots, p(n)]$.
   On utilisera un procédé récursif, basé sur la formule de récurrence vue en V-1-b.
   [Indication] [S]
4. Pour tout entier $n \geq 1$, on définit l'application $f_n : x \mapsto f_n(x) = \sum_{k=1}^{n} \begin{bmatrix} n \\ k \end{bmatrix} x^k$.
   (a) Montrer que $f_n(x) = x(x+1)\cdots(x+n-1)$. [S]
   (b) On choisit aléatoirement (avec équiprobabilité) une permutation p de $\mathcal{S}_n$.
   On note X_n la variable aléatoire représentant le nombre de cycles de p.
   Calculer l'espérance de X_n, et un équivalent de cette espérance quand $n \to \infty$. [S]
5. On va établir un résultat *apparemment* éloigné de ce qui précède...
   On considère tout d'abord un tableau aléatoire $Q = [a_1, a_2, \ldots, a_n]$ de n éléments distincts d'un ensemble totalement ordonné, par une relation notée $<$.
   (a) Écrire une fonction renvoyant le maximum des éléments du tableau Q. [S]
   (b) On dit qu'une valeur a_k de Q est un *record* si $k = 1$ ou si : $\forall j \in \{1, \ldots, k-1\},\ a_j < a_k$.
   Calculer l'espérance mathématique du nombre de records de Q... [S]

Corrigé avec Maple

I. Opérations sur les permutations

1. On forme un vecteur P de taille n, et on affecte gentiment chacune de ses composantes.

```
> identp:=proc(n::integer)
>   local P,k;
>   P:=array(1..n);              # ou encore :  P:=vector(n)
>   for k to n do P[k]:=k od;    # ou encore :  for k from 1 to n...
>   RETURN(eval(P))              # renvoie le tableau P
> end:
```

Voici la permutation *identité* dans $\mathcal{S}_7$:

```
> identp(7);
```

$$[1,2,3,4,5,6,7]$$

[Q]

2. On part de la permutation identité p représentée par le tableau $P=[1,2,\dots,n]$.
On échange $P[n]=n$ avec $P[j]=j$, où j est choisi aléatoirement dans $\{1,\dots,n-1\}$.
On échange ensuite $P[n-1]$ avec $P[j]$, où j est aléatoire dans $\{1,\dots,n-2\}$.
De proche en proche, on construit ainsi une permutation p aléatoire.

```
> randp:=proc(n::integer)
>   local P,j,k,t:  P:=identp(n);     # P = permutation identité
>   for k from n to 2 by -1 do        # pour k = n, n-1, ..., 2
>     j:=1+(rand()mod n);               # j ∈ [1,2,...,k], aléatoire
>     t:=P[k]; P[k]:=P[j]; P[j]:=t;     # échange P[k] et P[j]
>   od;
>   RETURN(eval(P));                  # renvoie la permutation aléatoire
> end:
```

On forme ici une permutation aléatoire de $\mathcal{S}_{10}$:

```
> randp(10);
```

$$[1,7,10,8,5,2,3,6,4,9]$$

Avec plus de liberté sur "`rand`" on peut remplacer "`rand()mod k`" par "`rand(k)()`".
On sait même que "`rand(j..k)()`" renvoie un entier aléatoire de $[j,\dots,k]$.
Avec cette syntaxe élargie, on peut légèrement simplifier la fonction "`randp`".

```
> randp:=proc(n::integer)
>   local P,j,k,t:
>   P:=identp(n);                   # P = permutation identité
>   for k to n-1 do                 # pour k = 1, 2, ..., n-1
>     j:=rand(k..n);                  # j ∈ [k,...,n], aléatoire
>     t:=P[k]; P[k]:=P[j]; P[j]:=t;   # échange P[k] et P[j]
>   od;
>   RETURN(eval(P));                # renvoie la permutation aléatoire
> end:
```

[Q]

3. Le résultat $t = q \circ p$ est représenté par un tableau T de taille n.
Pour chaque k de $\{1, \dots, n\}$ on a $t(k) = q(p(k))$ c'est-à-dire $T[k] = Q[P[k]]$.

```
> comp:=proc(Q::array(integer),P::array(integer),n::integer)
>   local T,k;
>   T:=array(1..n);                    # crée un tableau indicé de 1 à n
>   for k to n do T[k]:=Q[P[k]] od;    # calcule t = q ∘ p
>   RETURN(eval(T));                   # renvoie la permutation composée
> end:
```

Voici un exemple d'utilisation.

```
> P:=array([5,2,6,7,3,4,1]); Q:=array([1,6,5,4,3,2,7]); T:=comp(Q,P,7);
```

$$P := [5, 2, 6, 7, 3, 4, 1]$$
$$Q := [1, 6, 5, 4, 3, 2, 7]$$
$$T := [3, 6, 2, 7, 5, 4, 1]$$

[Q]

4. L'algorithme d'exponentiation rapide est basé sur l'écriture binaire de l'exposant m.
Posons en effet $m = \overline{b_k \dots b_1 b_0} = \sum_{j=0}^{k} b_j 2^j$ (pour tout j, $b_j = 0$ ou $b_j = 1$.)
Notons S l'ensemble des j de $\{0, \dots, k\}$ tels que $b_j = 1$. Alors $m = \sum_{j \in S} 2^j$ puis $p^m = \prod_{j \in S} p^{2^j}$.
Il suffit donc de calculer les $t_j = p^{2^j}$. Or $t_0 = p$ et pour tout j on a $t_{j+1} = t_j^2$.
On calcule les chiffres binaires b_j de m par divisions successives par 2 (le premier dividende est m, le suivant est le quotient entier de m par 2, etc.) Le chiffre b_j est le reste dans la $(j+1)$-ième division : j est dans S si le dividende de la $(j+1)$-ième division est impair.

(a) **Solution itérative** :

```
> powp:=proc(P::array(integer),n::integer,m::integer)
>   local Q,T,k;
>   Q:=identp(n);              # on initialise : q = Id
>   k:=m; T:=copy(P);          # on recopie m dans k et p dans t
>   while k>0 do               # tant que l'exposant résiduel k est > 0
>     if k mod 2 = 1 then        # ou if type(k,odd), ou  if irem(k,2)=1
>       Q:=comp(Q,T,n)           # si k est impair, on compose q et t
>     fi;
>     T:=comp(T,T,n);            # on élève t au carré
>     k:=floor(k/2);             # ou k:=iquo(k,2) : on divise k par 2
>   od;
>   RETURN(eval(Q));           # renvoie la puissance m-ième de p
> end:
```

Sur cet exemple, on calcule la puissance quatrième d'une permutation de $\mathcal{S}_9$:

```
> P:=array([8,6,1,3,4,5,2,9,7]); Q:=powp(P,9,4);
```

$$P := [8,6,1,3,4,5,2,9,7]$$
$$Q := [2,3,7,9,8,1,4,6,5]$$

[Q]

(b) **Solution récursive** :

On note que si $m \geq 1$, alors $p^m = \begin{cases} \left(p^{m/2}\right)^2 & \text{si } m \text{ est pair} \\ p \circ \left(p^{(m-1)/2}\right)^2 & \text{si } m \text{ est impair} \end{cases}$

Pour calculer $q = p^m$, on calcule donc $t = p^k$, avec $k = \mathrm{E}(\frac{m}{2})$, puis t^2.
On multiplie ensuite le résultat par p si l'exposant m est impair.
Évidemment, si $m = 0$, alors $p^m = \mathrm{Id}$ (c'est la condition d'arrêt de la plongée récursive.)
Pour la distinguer de la précédente, on a nommé `powpr` cette version récursive.

```
> powpr:=proc(P::array(integer),n::integer,m::integer)
>   local Q;
>   if m=0 then RETURN(identp(n))  # si m = 0 alors c'est l'identité.
>   else                           # si m ≥ 1,
>     Q:=powpr(P,n,iquo(m,2));       # appel récursif, avec exposant moitié
>     Q:=comp(Q,Q,n);                # élève le résultat au carré
>     if type(m,odd) then            # si m impair,
>       Q:=comp(P,Q,n)                 # on compose par p
>     fi;
>     RETURN(eval(Q));               # renvoie la puissance m-ième de p
>   fi
> end:
```

[Q]

5. La permutation p est représentée par le tableau $P = [p(1), p(2), \ldots, p(n)]$.
Soit $Q = [q(1), q(2), \ldots, q(n)]$ le tableau associé à $q = p^{-1}$.
Alors, pour tout entier k de $\{1, 2, \ldots, n\}$ on a $q(p(k)) = k$ donc $Q[P[k]] = k$.
Une simple boucle permet donc de placer les k successifs dans le tableau Q.

```
> invp:=proc(P::array(integer),n::integer)
>   local Q,k;
>   Q:=array(1..n);                # le tableau Q qui va représenter l'inverse de p
>   for k to n do Q[P[k]]:=k od;   # boucle d'affectation des composantes de Q
>   RETURN(eval(Q));               # renvoie la permutation inverse de p
> end:
```

Sur cet exemple, on calcule $q = p^{-1}$ et on vérifie que $q \circ p = \mathrm{id}$.

```
> P:=array([4,6,2,9,3,8,1,7,5]); Q:=invp(P,9); comp(Q,P,9);
```

$$P := [4,6,2,9,3,8,1,7,5]$$
$$Q := [7,3,5,1,9,2,8,6,4]$$
$$[1,2,3,4,5,6,7,8,9]$$

[Q]

II. Décomposition en cycles disjoints

1. Voici la procédure `Invp`, fidèle traduction des indications de l'énoncé.

```
> Invp:=proc(P::array(integer),n::integer)
>   local i,j,k,t:
>   for i to n do          # boucle de parcours du tableau
>     j:=i; k:=P[j];       # deux indices: k = p(j). Au départ j = i.
>     while P[i]>0 do      # tant que le cycle partant de i n'est pas traité
>       t:=P[k];           # sauvegarde l'image de k.
>       P[k]:=-j;          # écrit et "taggue" l'image réciproque p^{-1}(k) = j
>       j:=k;              # passe à l'élément qui suit j dans le cycle
>       k:=t               # récupère l'image k = p(j)
>     od;
>     P[i]:=-P[i];         # restaure le signe d'un l'élément "taggué"
>   od;
>   RETURN();              # aucun résultat renvoyé
> end:
```

Voici un exemple d'utilisation (avec la permutation p évoquée dans l'énoncé.)
On notera que le tableau représentant p est modifié par la procédure `Invp`.

```
> P:=array([7,2,5,8,1,4,3,6]): Invp(P,8); eval(P);
```

$$[5, 2, 7, 6, 3, 8, 1, 4]$$

[Q]

2. Voici la procédure `Decomp`.

```
> Decomp:=proc(P::array(integer),n::integer)
>   local Q,i,j,k;
>   Q:=copy(P);              # une copie de la permutation à décomposer
>   i:=0;                    # initialise le pointeur dans le tableau P
>   for j to n do            # boucle de parcours du tableau Q
>     while Q[j]>0 do          # tant que le cycle en cours n'est pas terminé,
>       i:=i+1: P[i]:=j;         # écrit l'élément courant dans le tableau P
>       j:=Q[j];                 # passe à l'élément suivant dans le cycle
>       Q[P[i]]:=0;              # marque comme lu le dernier élément traité dans Q
>       if Q[j]=0                # si le cycle est terminé,
>         then P[i]:=-P[i]         # marque la fin par un changement de signe
>       fi;
>     od;
>   od;
>   RETURN()                 # ne renvoie aucun résultat
> end:
```

Voici un exemple d'utilisation, avec la permutation citée dans l'énoncé.

```
P:=array([7,2,5,8,1,4,3,6]): Decomp(P,8); eval(P);
```

$$[1, 7, 3, -5, -2, 4, 8, -6]$$

Remarque : on constate que l'indice j de la boucle `for` subit des modifications dans le corps de cette boucle, à l'intérieur de la structure `while...do...od`. En fait, l'indice j décrit un cycle de la permutation, et il reprend à la sortie du `while` la valeur qu'il avait en y entrant. [Q]

3. Voici la procédure `Recomp`.
   Le tableau initial P est recopié dans une variable Q.
   Le tableau Q est ensuite parcouru sur toute sa longueur.
   Au fur et à mesure de ce parcours, la recomposition s'effectue dans le tableau P.
   On notera l'utilisation de la fonction `abs` pour les affectations dans P, de façon à neutraliser les changements de signe qui correspondent à des fins de cycle. On notera enfin l'utilisation de la variable locale d, dont le rôle est de mémoriser le début du cycle en cours de traitement.

```
> Recomp:=proc(P::array(integer),n::integer)
>   local Q,j,d;
>   Q:=copy(P);                  # une copie de la permutation à décomposer
>   for j to n do                # boucle de parcours du tableau Q
>     d:=Q[j];                      # c'est le début d'un cycle
>     while Q[j]>0 do               # tant que le cycle n'est pas terminé,
>       P[Q[j]]:=abs(Q[j+1]);          # complète la recomposition de la permutation
>       j:=j+1;                        # passe à l'élément suivant dans Q
>     od;
>     P[abs(Q[j])]:=abs(d);         # termine le traitement du cycle
>   od;
>   RETURN()                     # aucun résultat renvoyé
> end:
```

   Sur cet exemple, on décompose puis on recompose une permutation p.

```
> P:=array([7,2,5,8,1,4,3,6]); Decomp(P,8); eval(P); Recomp(P,8); eval(P);
```

$$P := [7, 2, 5, 8, 1, 4, 3, 6]$$
$$[1, 7, 3, -5, -2, 4, 8, -6]$$
$$[7, 2, 5, 8, 1, 4, 3, 6]$$

   On voit ici que `Recomp` rétablit la forme $\mathcal{T}$ d'une permutation à partir de l'une quelconque de ses formes $\mathcal{C}$ (et donc pas seulement à partir de celle qui a été fournie par `Decomp`.)

```
> P:=array([-2,8,6,-4,3,5,1,-7]); Recomp(P,8), eval(P);
```

$$P := [-2, 8, 6, -4, 3, 5, 1, -7]$$
$$[7, 2, 5, 8, 1, 4, 3, 6]$$

[Q]